Lektion 8

In dieser Lektion betrachten wir Heapsort, einen weiteren Sortieralgorithmus, der zunächst als sehr aufwendig und komplex erscheint, der sich aber dann bei der Komplexitätsanalyse als effizient herausstellt.

5.2 Heapsort

Alle bisher betrachteten Sortierverfahren, d.h. Insertionsort, Bubblesort und Selectionsort, haben im mittleren und im schlechtesten Fall eine Laufzeit in Größenordnung $\Theta(n^2)$. Zum Sortieren großer Datenmengen sind sie daher nicht geeignet.

In diesem Abschnitt betrachten wir *Heapsort*, ein Sortierverfahren, das darauf beruht, dass die Daten in einer speziellen Datenstruktur, genannt *Heap*, gespeichert werden. Im erstem Moment wirkt das Verfahren sehr kompliziert, bei der Analyse der Laufzeitkomplexität werden wird dann aber sehen, dass es tatsächlich ein effizientes Sortierverfahren ist.

5.2.1 Binäre Maximum-Heaps

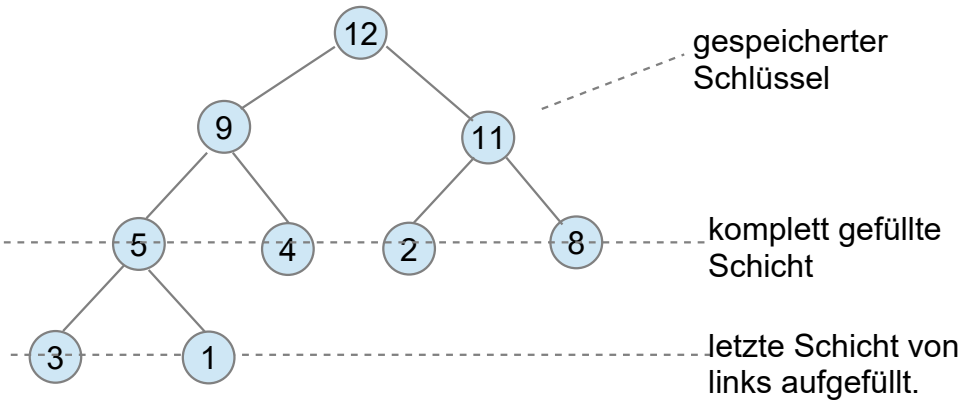
Ein Maximum-Heap ist eine baumartige Datenstruktur, die folgende Bedingungen erfüllt:

Definition 5.8 - Binärer Maximum-Heap

Ein **binärer Maximum-Heap** ist ein **binärer Baum,** der in jedem Knoten einen Schlüssel trägt und für den gilt:

- Der Baum ist **fast vollständig,** d.h. alle Ebenen bis auf die letzte sind komplett gefüllt. Die letzte Ebene ist lückenlos von links aus aufgefüllt.
- Für jeden Knoten gilt, dass der Schlüssel des Knotens **größer-gleich** der **Schlüssel seiner Kinder** ist.

Beispiel 5.9 - Binärer Maximum-Heap



Die Eigenschaften eines Maximum-Heaps sind hier erfüllt:

- Der Baum ist *fast vollständig*, d.h. bis zur vorletzten Schicht ist alles komplett aufgefüllt. Die letzte Schicht ist von links ohne Lücken gefüllt.
- Jeder Elternknoten ist größer als seine Kinder (gleiche Werte hat es hier im Beispiel nicht). Dies gilt nicht nur an der Wurzel, sondern auch für jeden anderen inneren Knoten.

Anmerkung

Der Begriff *Heap* hier hat nichts mit dem Begriff des Heaps tun, den Sie von der Speicherverwaltung bei Programmiersprachen, wie z.B. Java, kennen. Dort ist der Heap der Speicherbereich für dynamisch erzeugt Objekte und Arrays.

Feldrepräsentation für binäre Heaps

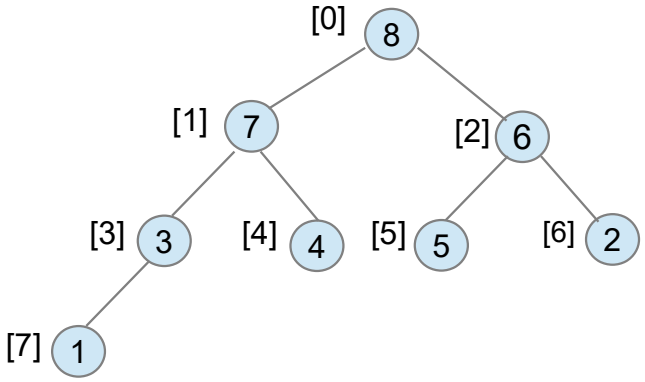
Ein fast vollständiger Binärbaum (und damit ein Heap s.o.) hat die angenehme Eigenschaft, dass der Baum sehr effizient in einem Array abgespeichert werden kann. Die Werte der Knoten werden Schichtweise von der Wurzel aus nacheinander im Feld abgelegt. Das lässt sich durch eine einfache Beziehung zwischen den Indizes von Eltern- und Kindknoten beschreiben:

Speicherung eines fast vollständigen Binärbaums im Array

- Die Wurzel wird an Index 0 abgelegt.
- Hat ein Knoten den Index **i**, dann haben die **Kinder** folgende Indizes :
  linkes Kind: **2i+1**
  rechtes Kind: **2i+2**

Beispiel 5.10 - Speicherung eines Heaps im Array

Dieser fast vollständige Binärbaum



würde also so in einem Array `a` abgespeichert werden:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
a:	8	7	6	3	4	5	2	1

Ein Vorteil dieser Repräsentation ist der, dass kein zusätzlicher Speicher benötigt wird, um die Eltern-Kind-Beziehung zu speichern.

Aufgabe 5.11 - Speicherung eines Heaps

Wie sieht der Baum aus, der in folgendem Array gespeichert ist?

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
8	7	6	6	1	5	1	4

Aufgabe 5.12 - Berechnung der Elternposition

Wie lässt sich für ein Kind, das an Index *i* im Array abgelegt ist, die Position des Elternknotens berechnen?

Da ein Heap einem vollständigen Binärbaum möglichst nahekommt, ergibt sich für die Höhe eines Heaps als Baum folgende Eigenschaft:

Eigenschaft 5.13 - Höhe eines Heaps

Ein fast vollständiger Binärbaum mit *n* Knoten (und damit ein Heap s.o.) hat eine Höhe von $\lceil \mathrm{ld}(n+1) \rceil$, d.h.

$$h(n) = \Theta(\log n).$$

Die Höhe hängt also also immer logarithmisch von der Anzahl der gespeicherten Werte ab.

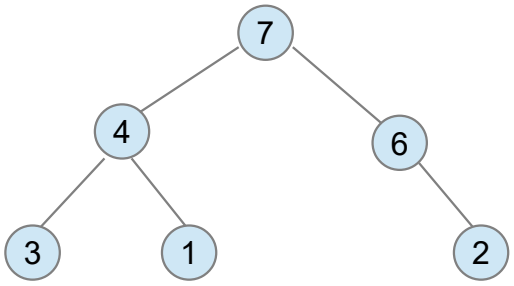
Anmerkungen

- Bei einem Maximum-Heap ist der Wert eines Knotens immer größer oder gleich aller Werte der darunter liegenden Teilbäume. Die **Wurzel** eines Maximum-Heaps, Element arr[0], enthält folglich das **Maximum** aller Werte im Baum (deswegen die Bezeichnung *Maximum-Heap*).
- Analog können binäre **Minimum-Heaps** definiert werden. Die Schlüssel der Elternknoten müssen dann jeweils kleiner-gleich der Schlüssel der Kinder sein. Wir werden Minimum-Heaps im nächsten Kapitel zur Implementierung von Prioritätswarteschlangen verwenden.

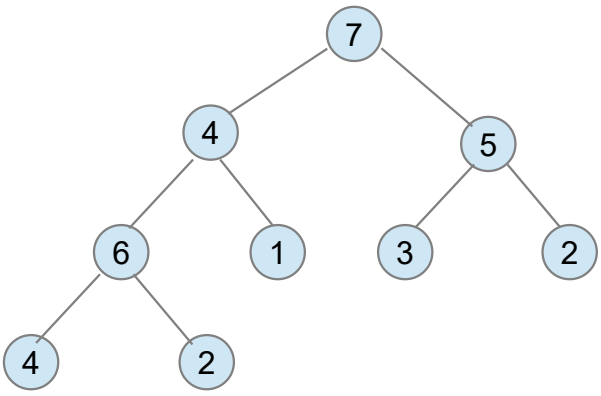
Aufgabe 5.14 - Binäre Maximum-Heaps

Welche der folgenden Bäume sind binäre Maximum-Heaps?

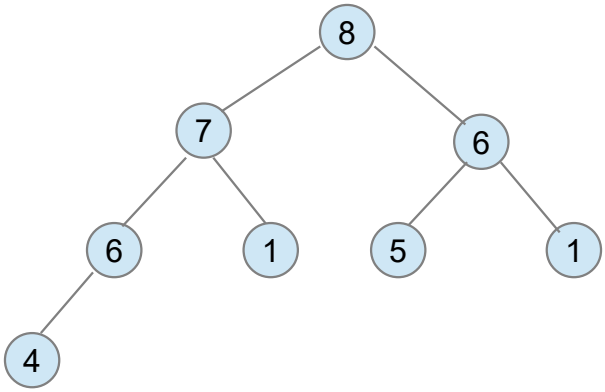
(1)



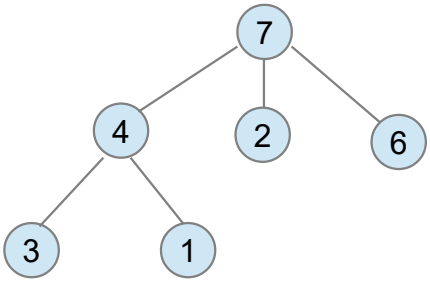
(2)



(3)



(4)



5.2.2 Sortieren mittels Heap

Was nützt ein Maximum-Heap, um Werte zu sortieren? Die wesentliche Eigenschaft, die zum Sortieren ausgenützt werden kann, ist die, dass bei einem Maximum-Heap der größte Wert an der Wurzel steht. Somit hat man einen Wert, dessen richtige Position fürs Sortieren bekannt ist. Danach müssen dann noch die restlichen Daten sortiert werden. Die Idee ist also ähnlich wie bei Selectionsort, mit dem Unterschied, dass die noch (weitgehend) unsortierten Daten in einem Heap verwaltet werden.

Allerdings gibt es zwei Herausforderungen auf dem Weg zu einem kompletten Sortierverfahren:

(1) Wie kommt man von den zunächst unsortierten Daten zu einem Maximum-Heap?

(2) Wenn aus einem Heap die Wurzel als größter Wert entfernt wurde, wie erhält man dann wieder einen Heap für die verbleibenden Daten, so dass anschließend der nächstkleinere Wert bestimmt werden kann?

Das Grundprinzip von Heapsort, das sich in zwei Phasen gliedert, lässt sich relativ einfach beschreiben. Die einzelnen Hilfsmethoden, die dafür benötigt werden, werden nachfolgend im Detail vorgestellt:

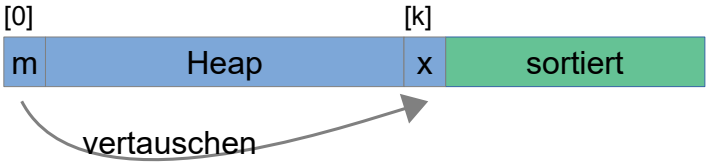
Grundprinzip von Heapsort

- **Phase 1 - Heap aufbauen**: Array a[0..n-1] mit unsortierten Daten wird in einen Maximum-Heap umgeordnet (Methode `erstelleMaxHeap`)
- **Phase 2 - Heap abbauen, sortierten Bereich aufbauen**: Gehe alle Positionen k absteigend von n-1 bis 1 (d.h. von hinten nach vorne) durch:
  1. Vertausche Wurzel a[0] und a[k]:
     Maximum a[0] aus dem Heap a[0..k] wird an Position k gebracht, um den sortierten Bereich damit zu vergrößern. Dadurch geht aber in der Regel die Heap-Eigenschaft an der Wurzel a[0] verloren
  2. Stelle Heap-Eigenschaft für a[0..k-1] wieder her durch "Versenken" von a[0] im verkleinerten Heapbereich a[0..k-1] (Methode `versenke`)

Am Ende bleibt a[0] als Heapbereich aus einem Element übrig. a[0] enthält dann den kleinsten Wert und steht damit schon an richtig Stelle.

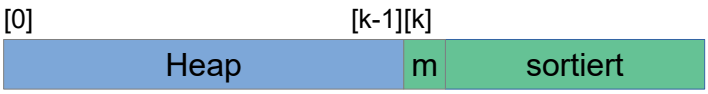
Anmerkung

- Die Grundidee für Phase 2 ist die, dass am Ende des Arrays ein Bereich mit den sortierten, größten Werten aufgebaut wird. Dieser Bereich wird in jedem Durchgang vergrößert, indem das Maximum des Heap-Bereichs davor (mit den noch nicht sortierten kleineren Werten) nach hinten getauscht wird.



Durch den Tausch kommt ein neues Element x an die Wurzel, so dass an der Wurzelposition die Heapeigenschaft nicht mehr erfüllt ist. Das Wurzelelement muss dann wieder in den Heap-Bereich passend eingefügt werden (mit Methode `versenke`)

Danach ist dann der nächste Durchgang möglich, in dem in gleicher Weise das nächstkleinere Element nach hinten einsortiert wird.



Eine Schleifeninvariante für Phase 2 von Heapsort ist also:

- Das gesamte Feld a[0..n-1] ist eine Permutation der Ausgangswerte
- Nach dem Durchgang zu Position *k* enthält der Bereich a[*k*..*n*-1] am Ende des Arrays aufsteigend sortiert die größten *n*-*k* Werte des Arrays (grüner Bereich oben).
- Der Bereich a[0..*k*-1] bildet jeweils einen Maximum-Heap (blauer Bereich oben). Alle Werte im Heap-Bereich sind kleiner-gleich der Werte im schon sortierten Bereich.

Eine zentrale Rolle bei Heapsort spielt folgende Methode `versenke`:

Algorithmus 5.15 - `versenke(a,i,m)`

Fügt den Wert a[i] in die darunter liegenden Teilbäume im Bereich *a*[*i* .. *m*] des Arrays ein, so dass die Heap-Eigenschaft wieder erfüllt ist.

Vorbedingung: Alle im Baum unter *a*[*i*] liegenden Knoten im Bereich *a*[*i*+1 ... *m*] erfüllen schon die Heap-Eigenschaft.

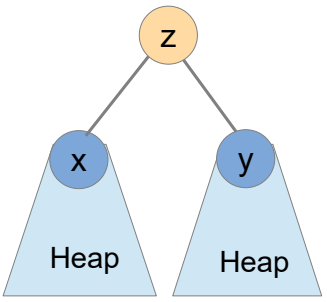
Vorgehensweise:

Setze *k* = *i* // *k* ist die aktuelle Indexposition des zu versenkenden Werts
wiederhole:
  falls Knoten an Position *k* kein Kind hat oder alle Kinder von *k* kleinere Werte haben
  dann
    fertig, Stop.
  sonst
    Sei *j* das Kind von *k* mit dem höchsten Wert.
    Vertausche die Werte an Position *k* und *j*;
    setze *k* = *j* // neue Position des zu versenkenden Werts

Nachbedingung: für alle Knoten/Werte im Bereich *a*[*i* ... *m*] gilt die Heap-Eigenschaft.

Das Vorgehen zum Versenken wird nachfolgend auf Baumebene veranschaulicht.

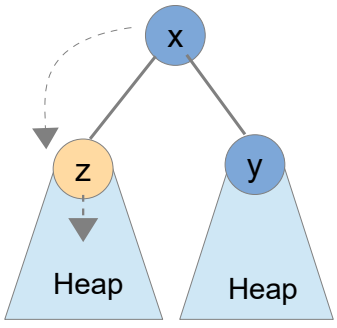
- Der Wert z soll in die darunterliegenden Teilbäume, die die Heap-Eigenschaft schon haben, eingefügt werden, so dass danach alles zusammen die Heap-Eigenschaft hat:



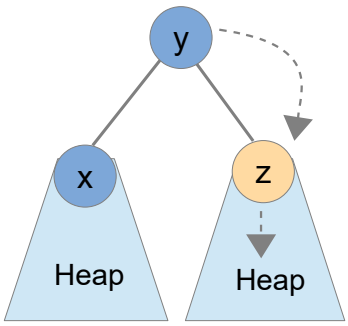
Es sind dann drei Fälle zu unterschieden:

- **Fall z ≥ x und z ≥ y**: Beide Kinder sind nicht größer, z ist somit schon an richtiger Stelle, fertig!
- **Fall z < x und x ≥ y**: Ist x das Maximum der Kinder und ist x größer als z, dann wird x nach oben getauscht und z muss weiter in den linken Teilbaum eingefügt werden.

  Da x größer als z ist und größer als y ist, ist für x die Position Heapeigenschaft erfüllt, dass der Wert größer-gleich der Kinderwerte ist. z muss ggf. noch weiter im linken Teilbaum versenkt werden.

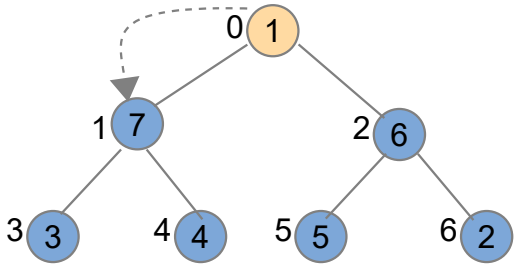


- **Fall z < y und y ≥ x**: Ist y das Maximum der Kinder und ist y größer als z, dann wird y nach oben getauscht und z muss in den rechten Teilbaum eingefügt werden.
  Da ~~x~~ (y) größer als z und als ~~y~~ (x) ist, ist für y die Heapeigenschaft erfüllt. z muss aber ggf. noch weiter im rechten Teilbaum versenkt werden.

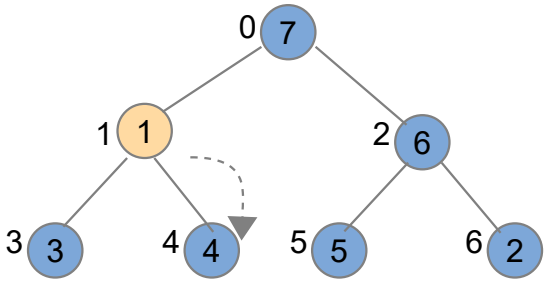


Beispiel 5.16 - Operation versenke()

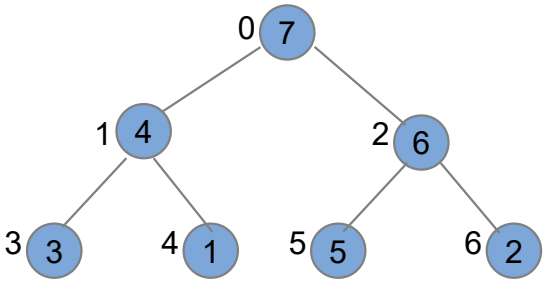
Wurzelwert 1 ist zu versenken, die linken und rechten Teilbäume haben schon Maximum-Heap-Eigenschaft:



Linkes Kind mit Wert 7 ist Maximum der Kinder und größer als 1. Also wird 7 mit 1 vertauscht.



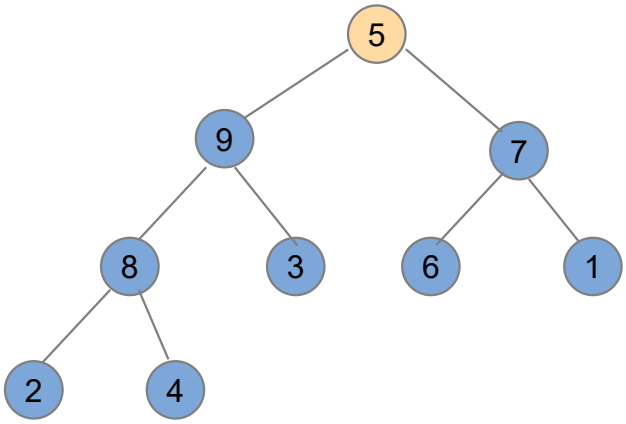
Wert 1 muss nun ab der neuen Position weiter versenkt werden. Das rechte Kind mit Wert 4 ist das größere der beiden Kinder und größer als 1. Also wird 1 nach rechts mit 4 getauscht.



Da 1 nun ein Blatt ist, endet das Versenken. Der Baum hat jetzt komplett die Maximumheap-Eigenschaft.

Aufgabe 5.17 - Operation versenke()

Gegeben ist folgender fast vollständige Binärbaum, bei dem bereits alle Knoten außer der Wurzel 5 die Heap-Eigenschaft erfüllen.



(1) Wie würde der Baum in einem Array gespeichert werden?

(2) Versenken Sie den Wurzelwert 5:

Nun sollte auch folgende Java-Implementierung für die Operation `versenke()` verständlich sein.

- In Variable `k` wird dabei die aktuelle Indexposition des zu versenkenden Elements gespeichert.
- Mit der Bedingung `k <= (m / 2) - 1` der Schleife wird geprüft, ob der aktuell behandelte Knoten in der linken Hälfte des Heapbereichs ist und folglich ein innerer Knoten mit mindestens einem Kind ist.

Algorithmus 5.18 - Java-Implementierung von versenke()

```
public static void versenke(double[] a, int i, int m) {
  int k = i;
  while (k <= (m / 2) - 1) {
        // in linker Hälfte, kein Blatt, mindestens ein Kind existiert
      int kindLinks = 2 * k + 1;
      int kindRechts = kindLinks + 1;

      //bestimme Kind mit größtem Wert
      int maxKind = kindLinks;
      // bestimme ob ein rechtes Kind existiert und einen größeren Wert hat
      if (kindRechts <= m - 1 &&
            a[kindLinks] < a[kindRechts]) {
          maxKind = kindRechts;
```

```
        }

        // prüfe, ob Element weiter versenkt werden muss
        if (a[k] < a[maxKind]) {
            // Maximum der Kinder ist größer,
            // Element durch Tausch eine Stufe tiefer sinken lassen
            vertausche(a, k, maxKind);
            k = maxKind;      /
            // wiederhole den Vorgang mit der neuen Position
        } else {
            // Element ist an passender Position gelandet, fertig!
            break;
        }
    }
}
```

Methode `versenke()` ist der kniffligste Teil von Heapsort. Alles weitere kann nun einfach implementiert werden.

Phase 1 - Aufbauen des Heaps (`erstelleMaxHeap`)

Die unsortierten Daten im Array a[0..*n*-1] müssen effizient zu einem Maximum-Heap umgeordnet werden. Der Feldinhalt kann schon als ein fast vollständiger Binärbaum betrachtet werden. Allerdings ist die Forderung, dass Eltern größer-gleich Kinder sein müssen, noch nicht erfüllt.

Maximum-Heap erstellen

Die Heap-Eigenschaft wird "bottom-up", d.h. von den Blättern ausgehend, hergestellt:

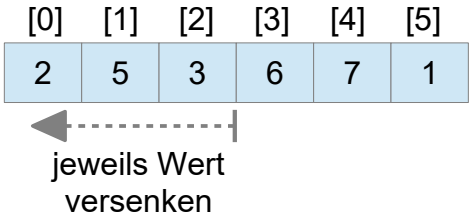
- Alle Blätter erfüllen die Eigenschaft, da sie keine Kinder haben. Bei einem fast vollständigen Binärbaum sind die Blätter immer genau die rechte Hälfte a[$\lfloor n/2 \rfloor$ .. *n*-1] des Arrays der Länge *n*.
- Die Elemente a[i] der linken Hälfte a[0..$\lfloor$n/2$\rfloor$-1] des Arrays werden *von der Mitte aus rückwärts zum Array-Anfang, also von rechts nach links* durchgegangen (und damit im Baum auch von unten nach oben Richtung Wurzel) und es wird die Operation `versenke` auf das aktuelle Element a[i] angewendet, um das Element in die darunter liegenden Teilbäume (die schon die Heap-Eigenschaft haben) so einzufügen, dass der ganze Teilbaum die Heapeigenschaft erfüllt

Algorithmus 5.19 - Java-Implementierung erstelleMaxHeap()

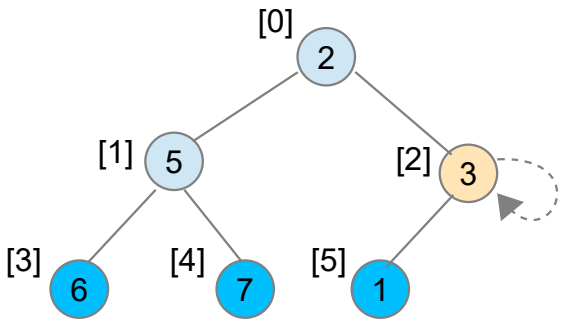
```
private static void erstelleMaxHeap(double[] a) {
  for (int i = (a.length / 2) - 1; i >= 0; i--) {
      versenke(a, i, a.length);
  }
}
```

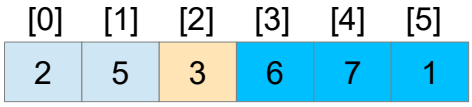
Beispiel 5.20 - erstelleMaxHeap()

Folgendes Array soll mit `erstelleMaxHeap()` in einen Maximum-Heap umgeordnet werden:

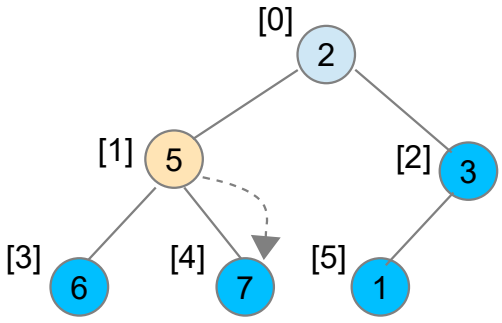


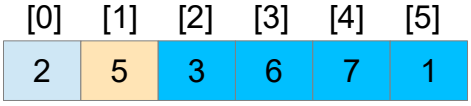
(1) Wert 3 an Position [2] versenken:



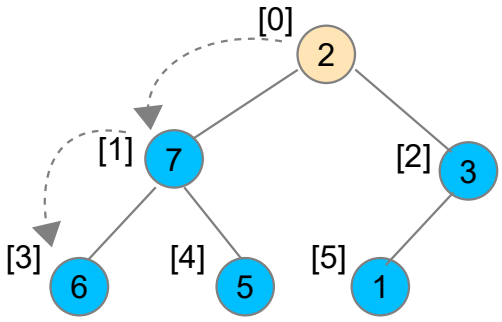


(2) Wert 5 an Position [1] versenken:



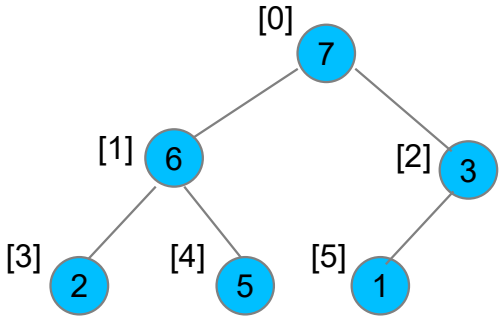


(3) Wert 2 an Position [0] versenken:



[0]	[1]	[2]	[3]	[4]	[5]
2	7	3	6	5	1

Endergebnis:



[0]	[1]	[2]	[3]	[4]	[5]
7	6	3	2	5	1

Aufgabe 5.21 - erstelleMaxHeap()

Folgendes Array soll mit `erstelleMaxHeap()` zu einem Maximum-Heap umgeordnet werden:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
3	7	5	8	4	6	2	1

Mit den beiden Hilfsmethoden `versenke` und `erstelleMaxHeap` lässt sich nun Heapsort in wenigen Programmzeilen formulieren:

Algorithmus 5.22 - Java-Implementierung für Heapsort

```
public static void heapSort(double[] a) {

  // Phase 1 - aus dem Ausgangsfeld einen Maximum-Heap aufbauen:
  erstelleMaxHeap(a);

  // Phase 2 - Heap nach und nach wieder abbauen, um die sortierte Folge am
  // Ende des Arrays aufzubauen:
  for (int i = a.length - 1; i > 0; i--) {
      //größtes Element aus dem Heap entfernen und den sortierten Bereich
      //damit erweitern, d.h. Elemente a[0] mit a[i] vertauschen
     vertausche(a, 0, i);
```

```
        //Heap-Eigenschaft für den Bereich a[0..i-1] wieder herstellen
        versenke(a, 0, i);
    }
}
```

Die verwendete Hilfsmethode `vertausche(arr, j, k)` vertauscht die Einträge in einem Array `arr` an den Positionen `j` und `k`.

Beispiel 5.23 - Heapsort

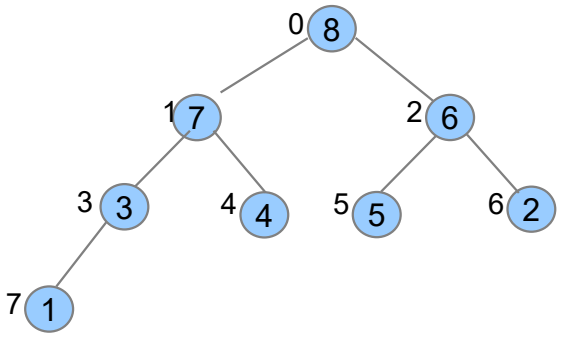
Das Array

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
3	7	5	8	4	6	2	1

soll mit Heapsort sortiert werden.

Phase 1: Heap aufbauen mit erstelleMaxHeap():

- siehe Aufgabe 5.21 - erstelleMaxHeap(), liefert folgendes Ergebnis:

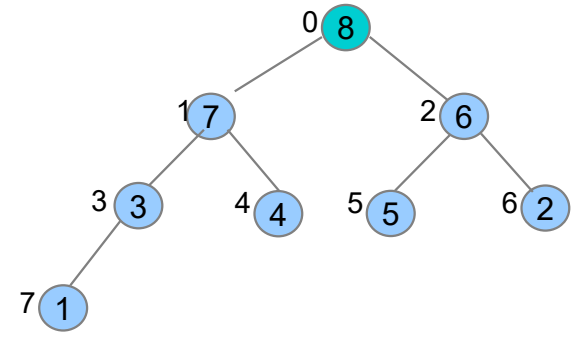


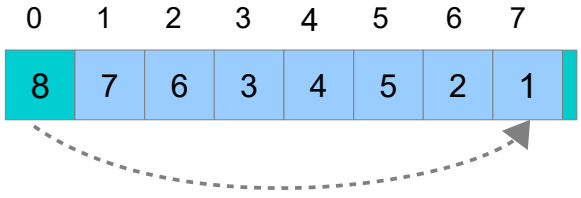
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
8	7	6	3	4	5	2	1

Phase 2: Heap nach und nach abbauen, sortierten Bereich aufbauen

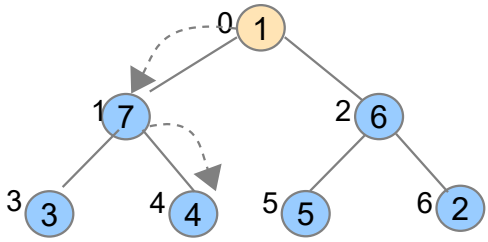
- Ausgangszustand: sortierter Bereich am Ende (grün) noch leer, der Maximum-Heap umfasst das komplette Array.

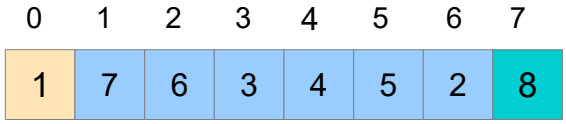
  Maximum 8 an Wurzelposition a[0] nach hinten tauschen:



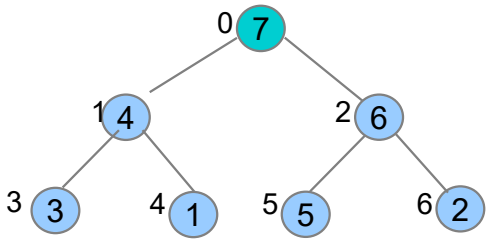


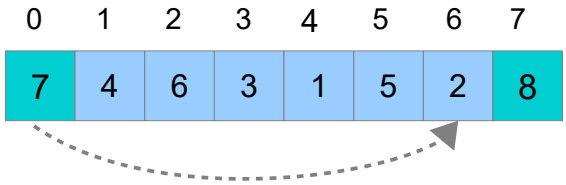
Wurzelwert 1 versenken, um Heapeigenschaft wieder herzustellen, so dass Maximum der noch unsortierten Werte wieder an Wurzelposition ist.



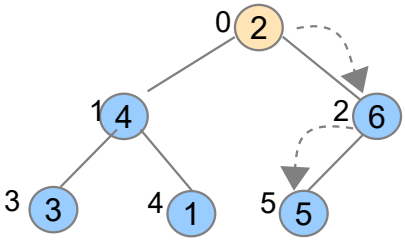


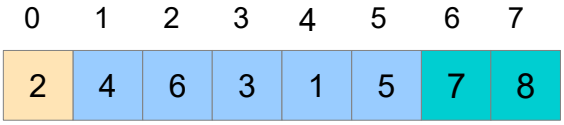
- Maximum 7 des Heapbereichs nach hinten tauschen:



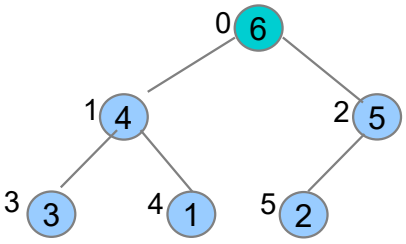


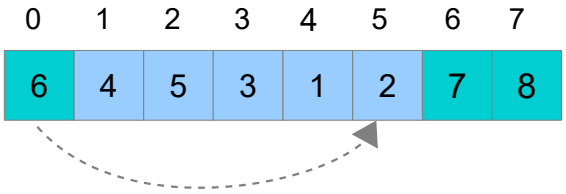
Wert 2 an Wurzel versenken, um Heap-Eigenschaft wieder herzustellen.



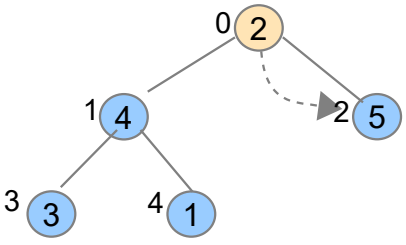


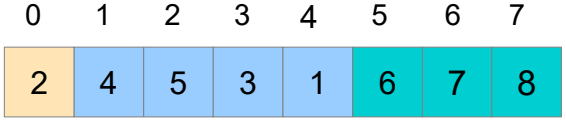
- Maximum 6 des Heapbereichs nach hinten tauschen. 2 kommt an Wurzel.



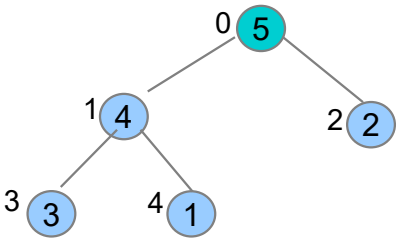


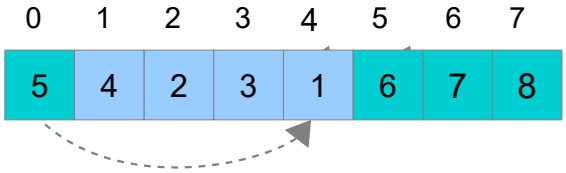
Wurzel 2 versenken, um Heap-Eigenschaft wieder herzustellen:



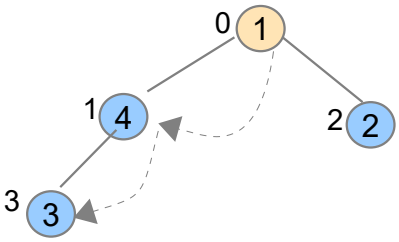


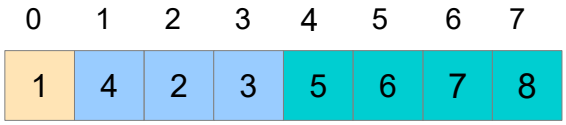
- Maximum 5 nach hinten tauschen:



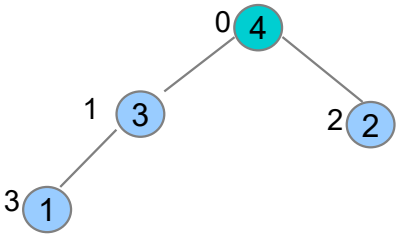


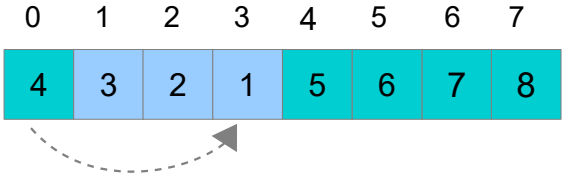
1 versenken, um Heapeigenschaft wieder herzustellen.



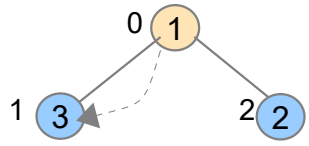


- Maximum 4 des Heaps nach hinten tauschen:

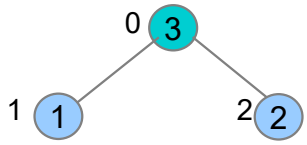


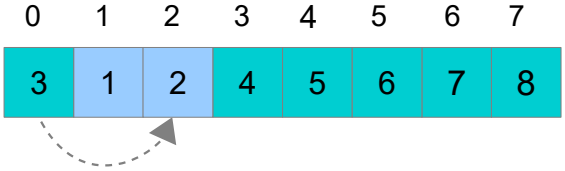


Wurzelwert 1 versenken:

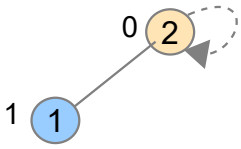


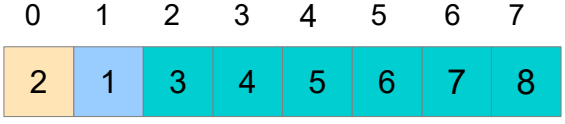
- Maximum 3 nach hinten tauschen:



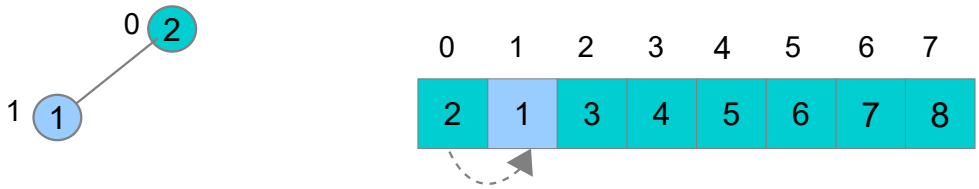


Wurzelwert 2 versenken:

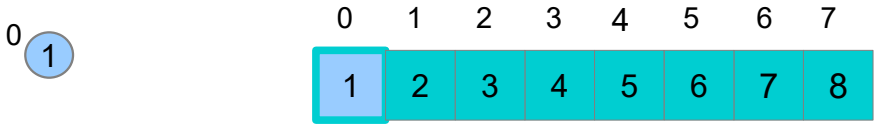




- Maximum 2 nach hinten tauschen.



- Ein Wert bleibt noch im Heap-Bereich, der kleinste aller Werte. Er ist somit schon an richtiger Stelle. Phase 2 ist damit beendet, das Array ist komplett sortiert.



5.2.3 Laufzeitkomplexität von Heapsort

Heapsort erscheint Ihnen zunächst vermutlich sehr kompliziert und aufwendig. Kann es trotzdem ein effizientes Sortierverfahren sein? Wir untersuchen dazu das Verhalten im schlechtesten Fall (worst case).

Zunächst betrachten wir die Operation `versenke`, die die zentrale Rolle bei Heapsort spielt:

Laufzeitkomplexität für Operation `versenke`

Die Anzahl der erforderlichen Vergleiche und Vertauschungen ist durch die Höhe des Heaps als Baum beschränkt, da beim Versenken im schlechtesten Fall das Element von der Wurzel bis zum Blatt mit maximaler Tiefe getauscht werden muss. Da der Heap ein fast vollständiger Binärbaum ist, ist die Höhe ca. ld(n), d.h. $\Theta(\log n)$. Folglich ist auch die Laufzeitkomplexität, um einen Wert in einem Teilbaum mit n Knoten zu versenken, logarithmisch:

$$T_{versenke}(n) = O(\log n)$$

Laufzeitkomplexität für Operation `erstelleMaxHeap`

Bei `erstelleMaxHeap` werden alle inneren Knoten versenkt. Das ist bei einem fast vollständigen Binärbaum die Hälfte der Elemente. Wir schätzen großzügig ab, dass die Teilbäume, in die die Elemente versenkt werden, immer maximal n Knoten haben und somit die einzelne `versenke`-Operation jeweils den Aufwand $O(\log n)$ hat.

$$T_{erstelleMaxHeap}(n) = n/2 \cdot T_{versenke}(n) = n/2 \cdot O(\log n) = O(n \log n)$$

Laufzeitkomplexität für Heapsort

- **Phase 1 von Heapsort** ist die Operation `erstelleMaxHeap`, somit ist die Laufzeitkomplexität dafür
  $O(n \log n)$
- **Phase 2 von Heapsort**: Es wird n-1 mal wiederholt:

  (1) Maximum an Wurzelposition nach hinten tauschen: $O(1)$

  (2) Wurzelelement versenken: Wir schätzen hier wieder konservativ ab, dass der Wert jeweils in einen Teilbaum von maximal n Elementen versenkt werden muss. D.h. der Aufwand ist in Größenordnung

  $(n\text{-}1)(O(1) + O(\log n)) = O(n \log n)$

Insgesamt ergibt sich somit im **schlechtesten Fall** (Phase 1 + Phase 2):
$T_{Heapsort}(n) = O(n \log n) + O(n \log n) = O(n \log n)$

Wir haben verschiedene Abschätzungen vorgenommen. Dadurch habe wir gezeigt, dass $O(n \log n)$ eine Obergrenze ist. Durch genauere Analysen kann gezeigt werden, dass $n \log n$ auch die Untergrenze ist, so dass die Laufzeitkomplexität genau in Größenordnung $n \log n$ liegt, d.h. $\Theta(n \log n)$ ist.

Es kann auch gezeigt werden, dass dies nicht nur im schlechtesten Fall so ist, sondern auch im besten und im mittleren Fall.

Laufzeitkomplexität von Heapsort
$\mathbf{T_{best}(n) = T_{avg}(n) = T_{worst}(n) = \Theta(n \log n)}$

Im mittleren und schlechtesten Fall ist Heapsort mit $\Theta(n \log n)$ also besser als die bisher betrachteten einfachen Sortierverfahren Insertionsort, Bubblesort und Selectionsort.

Sie werden in einer der kommenden Übungsaufgaben die Laufzeiten von Sortierverfahren experimentell messen und dann selbst erleben, wie sich der Unterschied zwischen einem effizienten Verfahren mit $\Theta(n \log n)$ und einem nicht effizienten Verfahren mit $\Theta(n^2)$ bemerkbar macht.

Aufgabe 5.24 - Warum Maximum-Heap?

Bei Heapsort wird ein Maximum-Heap verwendet und der sortierte Bereich am Array-Ende von den größten Elementen aus ausgehend aufgebaut.

Wäre es auch möglich, mit einem Minimum-Heap zu arbeiten und von vorne mit den kleinsten Elementen anzufangen?

Fazit zu Lektion 8

Das sollten Sie in dieser Lektion gelernt haben

- Was ist ein binärer Maximum-Heap?
- Wie kann ein fast vollständiger Binärbaum effizient in einem Array abgespeichert werden? Wie können dabei die Positionen von Kindern bzw. Eltern berechnet werden?
- Wie arbeitet Heapsort? In welche zwei Hauptphasen gliedert sich der Heapsort-Algorithmus?
- Wie kann effizient aus unsortierten Daten ein binärer Maximum-Heap erstellt werden?
- Wie kann durch die versenke-Operation die Heap-Eigenschaft wieder hergestellt werden?
- Welche Laufzeitkomplexität hat Heapsort?